Third International Conference on Generative and □
Component-Based Software Engineering (GCSE 2001), □
September 9-13, 2001 Messe Erfurt, Erfurt, Germany.

A Standard Problem for Evaluating Product-Line Methodologies

Roberto E. Lopez-Herrejon and Don Batory
Department of Computer Sciences
The University of Texas
Austin, Texas 78712
{rlopez,batory}@cs.utexas.edu

Abstract. We propose a standard problem to evaluate product-line methodologies. It relies on common knowledge from Computer Science, so that domain-knowledge can be easily acquired, and it is complex enough to expose the fundamental concepts of product-line methodologies. As a reference point, we present a solution to this problem using the GenVoca design methodology. We explain a series of modeling, implementation, and benchmarking issues that we encountered, so that others can understand and compare our solution with theirs.

1 Introduction

A *product-line* is a family of related software applications. A *product-line architecture* is a design for a product-line that identifies the underlying building blocks or *components* of family members, and enables the synthesis of any particular member by composing these components. Different family members (product-line applications) are represented by different combination of components. The motivation for product-line architectures is one of economics and practicality: it is too expensive to build all possible family members; it is much cheaper to build components and to assemble desired family members from them.

Many methodologies have been invented to create product-line architectures (e.g., [2, 3, 7, 9, 11, 12, 13, 14, 17, 20]). Unfortunately, the state-of-the-art is immature. We are unaware of any attempts to evaluate different methodologies on a common set of problems. If this were done, we would understand better the strengths and weaknesses of different methodologies. We would know when to use a particular methodology, and when not to. Further, we would know if different methodologies relied on the same concepts. For example, different OO design approaches rely on a common conceptual foundation of classes, interfaces, and state machines, but offer different ways of producing a design expressed in terms of these concepts. For product-line methodologies, we generally do not know even this. Different methodologies have rather different meanings for the terms “architecture”, “component”, and “composition” so that it is not at all obvious what, if anything, is in common. It is not evident that the same concepts are shared among product-line methodologies, let alone knowing what these concepts are. From a practical standpoint, the choice of which methodology to use in a situation is dictated by convenience (at best) or by random selection (at worst) rather than by scientific fact. This is unacceptable.

For this area to mature, it is essential that we compare and evaluate proposed methodologies. The scientific principles that underlie this area must be identified and the contributions and novelties of different methodologies be exposed in a way that all can appreciate and recognize. The immaturity of this area is not unique and has occurred in other areas of Computer Science. In such cases, a standard problem has been proposed and different authors have applied their methodologies to solve it (e.g., [1]). Doing so exposes important details that would otherwise be overlooked or misunderstood. Such studies allow researchers to more accurately assess the strengths, benefits, commonalities, and variabilities of different methodologies. We believe this approach would be beneficial for product-lines.

In this paper, we propose a standard problem for evaluating product-line methodologies. We believe a standard problem should have the following characteristics:

- It draws on common knowledge from Computer Science, so that the often difficult requirement of becoming a domain expert or acquiring domain-expertise is minimized.
- It is not a trivial design problem; it is complex enough to expose the key concepts of product-lines and their implementation.

These characteristics should enable others to see the similarities and differences among approaches both at a superficial level and more importantly, at a deeper conceptual level.

To carry this idea forward, we present as reference point a solution to this problem using the GenVoca design methodology. We outline a set of design, implementation, and benchmarking issues that we had to resolve before we settled on our final design. Doing so exposed a variety of concerns and insights that we believe others would benefit hearing. Our designs, code, and benchmarks are available at a web site (`http://www.cs.utexas.edu/users/dsb/GPL.html`) for others to access.

2 A Standard Problem: The Graph Product Line

The *Graph Product-Line (GPL)* is a family of classical graph applications that was inspired by early work on software extensibility [16, 19]. GPL is typical of product-lines in that applications are distinguished by the set of features that they implement, where no two applications have the same set of features.[1] Further, applications are modeled as sentences of a grammar. Figure 1a[2] shows this grammar, where tokens are names of features. Figure 1b shows a GUI that implements this grammar and allows GPL products to be specified declaratively as a series of radio-button and check-box selections.

1. A *feature* is a functionality or implementation characteristic that is important to clients [15].
2. For simplicity, the grammar does not preclude the repetition of algorithms, whereas the GUI does.

(a)

```
GPL := Gtp Wgt Src Alg+;

Gtp := Directed | Undirected;

Wgt := Weighted | Unweighted;

Src := DFS | BFS | None;

Alg := Number | Connected | StronglyConnected
     | Cycle | MST Prim | MST Kruskal | Shortest;
```

(b)

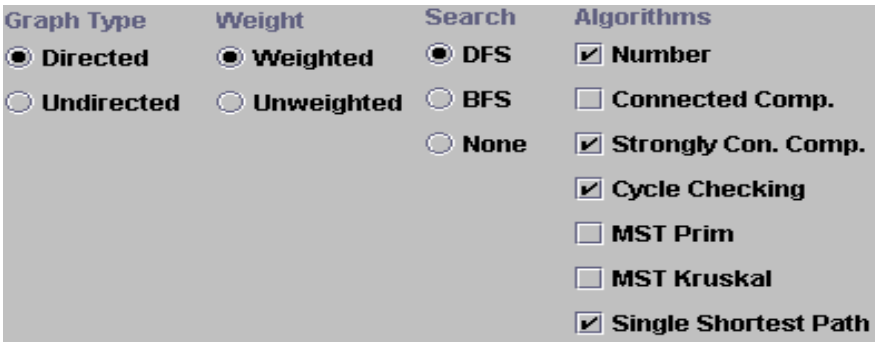


Figure 1. GPL Grammar and Specification GUI

The semantics of GPL features, and the domain itself, are uncomplicated. A graph is either `Directed` or `Undirected`. Edges can be `Weighted` with non-negative numbers or `Unweighted`. Every graph application requires at most one search algorithm: breadth-first search (`BFS`) or depth-first search (`DFS`); and one or more of the following algorithms [10]:

- **Vertex Numbering (`Number`):** Assigns a unique number to each vertex as a result of a graph traversal.
- **Connected Components (`Connected`):** Computes the *connected components* of an undirected graph, which are equivalence classes under the reachable-from relation. For every pair of vertices x and y in an equivalence class, there is a path from x to y.
- **Strongly Connected Components (`StronglyConnected`):** Computes the *strongly connected components* of a directed graph, which are equivalence classes under the reachable-from relation. A vertex y is reachable form vertex x if there is a path from x to y.
- **Cycle Checking (`Cycle`)**: Determines if there are cycles in a graph. A cycle in directed graphs must have at least 2 edges, while in undirected graphs it must have at least 3 edges.

- **Minimum Spanning Tree (`MST Prim`, `MST Kruskal`):** Computes a *Minimum Spanning Tree (MST)*, which contains all the vertices in the graph such that the sum of the weights of the edges in the tree is minimal. We include both algorithms because they present distinct and interesting performance and design issues.
- **Single-Source Shortest Path (`Shortest`):** Computes the shortest path from a source vertex to all other vertices.

A fundamental characteristic of product-lines is that not all features are compatible. That is, the selection of one feature may disable (or enable) the selection of others. GPL is no exception. The set of constraints that govern the GPL features are summarized in Table 1.

Algorithm	**Required Graph Type**	**Required Weight**	**Required Search**
Vertex Numbering	Directed, Undirected	Weighted, Unweighted	BFS, DFS
Connected Components	Undirected	Weighted, Unweighted	BFS, DFS
Strongly Connected Components	Directed	Weighted, Unweighted	DFS
Cycle Checking	Directed, Undirected	Weighted, Unweighted	DFS
Minimum Spanning Tree	Undirected	Weighted	None
Single-Source Shortest Path	Directed	Weighted	None

Table 1. Feature Constraints

A GPL application implements a valid combination of features. As examples, one GPL application implements vertex numbering and connected components using depth-first search on an undirected graph. Another implements minimum spanning trees on weighted, undirected graphs. Thus, from a client's viewpoint, to specify a particular graph application with the desired set of features is straightforward. And so too is the implementation of the GUI (Figure 1b) and constraints of Table 1.

We chose Java as our implementation language. Besides its simplicity over C++ and availability of GUI libraries, we made use of Java containers, iterators, and sort methods, to avoid reimplementing these low-level routines by hand. We recommend others to follow our lead to make comparisons easier.

3 GenVoca

GenVoca is a model of product-lines that is based on step-wise extension [3-6][3]. Among its key ideas is programs are values. Consider the following constants that represent programs with individual features:

```
f          // program that implements feature f
g          // program that implements feature g
```

An *extension* is a function that takes a program as input and produces an extended (or feature-augmented) program as output:

```
i(x)       // adds feature i to program x
j(x)       // adds feature j to program x
```

It follows that a multi-featured application is an *equation*, and that different equations define a set of related applications, i.e., a *product-line*, such as:

```
a1 = i(f)       // application a1 has features i and f
a2 = j(g)       // application a2 has features j and g
a3 = i(j(f))    // application a3 has features i, j, and f
```

Thus one can determine features of an application by inspecting its equation.

3.1 GPL

A GenVoca model of GPL is the set of constants and functions defined in Table 2. There are three extensions that are not visible to the GUI: `Transpose`, `Benchmark`, and `Prog`. `Transpose` performs graph transposition and is used (only) by the `StronglyConnected` algorithm. It made sense to separate the `StronglyConnected` algorithm from `Transpose`, as they dealt with separate concerns. (This means that an implementation constraint in using the `StronglyConnected` extension is that the `Transpose` extension must also be included, and vice versa). `Benchmark` contains functions to read a graph from a file and elementary timing functions for profiling. `Prog` creates the objects required to represent a graph, and calls the algorithms of the family member on this graph.

Extensions can not be composed in arbitrary orders. The legal compositions of extensions in Table 2 are defined by simple constraints called *design rules* [3] whose details we omit from this paper, but do include with our source code. Our GUI specification tool translates a sentence in the grammar of Figure 1 (in addition to checking for illegal combinations of features) into an equation. Because features are in 1-to-1 corre-

3. A *refinement* adds implementation detail, but does not add methods to a class or change the semantics of existing methods. In contrast, *extensions* not only add implementation detail but also can add methods to a class and change the semantics of existing methods. Inheritance is a common way to extend classes statically in OO programming languages.

spondence with extensions, this translation is straightforward. For example, a GPL application `app` that implements vertex numbering and connected components using depth-first search on an undirected graph is the equation:

```
app = Prog( Benchmark( Number( Connected( DFS( Undirected )))))
```

`Directed`	directed graph	`Cycle(x)`	cycle checking
`Undirected`	undirected graph	`MSTPrim(x)`	MST Prim algorithm
`Weighted(x)`	weighted graph	`MSTKruskal(x)`	MST Kruskal algorithm
`DFS(x)`	depth-first search	`Shortest(x)`	single source shortest path
`BFS(x)`	breadth-first search	`Transpose(x)`	graph transposition
`Number(x)`	vertex numbering	`Benchmark(x)`	benchmark program
`Connected(x)`	connected components	`Prog(x)`	main program
`StronglyConnected(x)`	strongly connected components		

Table 2. A GenVoca Model of GPL

3.2 Mixin-Layers

There are many ways in which to implement extensions. We use mixin-layers [18]. To illustrate, recall the `Directed` program implements a directed graph. This program is defined by multiple classes, say `Graph`, `Vertex`, and `Edge`. (The exact set of classes is an interesting design problem which we discuss in Section 4). A mixin-layer that represents the `Directed` program is the class `Directed` with inner classes `Graph`, `Vertex`, and `Edge`:

```
class Directed {
   class Graph  {...}
   class Vertex {...}
   class Edge   {...}
}
```

An extension is implemented as a mixin, i.e., a class whose superclass is specified by a parameter. The depth-first search extension is implemented as a mixin `DFS` that encapsulates extensions (mixins) of the `Graph` and `Vertex` classes. That is, `DFS` grafts new methods and variables onto the `Graph` and `Vertex` classes to implement depth first search algorithms:

```
class DFS<x> extends x {
   class Graph  extends x.Graph  {...}
   class Vertex extends x.Vertex {...}
}
```

The above describes the general way in which GenVoca-GPL model constants and functions are implemented. When we write the composition `A = DFS(Directed)` in our model, we translate this to the equivalent template expression:

```
class A extends DFS<Directed>;
```

In general, there is a simple mapping of model equations to template/mixin expressions. Of course, Java does not support mixins or mixin-layers, but extended Java languages do. We used the *Jakarta Tool Suite (JTS)* to implement mixin-layers [4].

4 Graph Implementation

Designing programs that implement graph algorithms is an interesting problem. Every implementation will define a representation of graphs, vertices, edges, and *adjacency* — i.e., what vertices are adjacent (via an edge) to a given vertex. Further, there must be some way to represent annotations of edges (e.g., weights, names). We did not arrive at our final design immediately; we went through a series of designs that incrementally improved the clarity of our code, which we document in the following sections. In the process, we learned a simple rule to follow in order to simplify extension-based designs.

4.1 Adjacency Lists Representation (G)

The first representation we tried was based on a "legacy" C++ design [18, 5] that had been written years earlier and that implemented few of the extensions listed in Table 2. It consisted of 2 classes:

- `Graph`: consists of a list of `Vertex` objects.
- `Vertex`: contains a list of its adjacent `Vertex` objects.

That is, edges were implicit: their existence could be inferred from an adjacency list. Figure 2 illustrates this representation for a weighted graph. The advantage of this representation was its simplicity. It worked reasonably well for most extensions that we had to implement. However, it failed on edge annotations (e.g., weights). Because edges were implicitly encoded in the design, we had to maintain a weights list that was "parallel" to the adjacency list. While this did indeed work, our layered designs were obviously not clean or elegant — e.g., for operations like graph transposition which needed to read edge weights, and Kruskal's algorithm which needed to manipulate edges directly. Because of these reasons, this lead us to our second design.

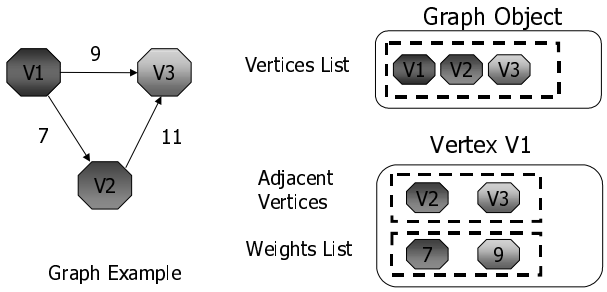


Figure 2. Adjacency Lists Representation Example

4.2 Neighbor List Representation (GN)

The second representation consisted of three classes:

- `Graph`: contains a list of `Vertex` objects.
- `Vertex`: contains a list of `Neighbor` objects.
- `Neighbor`: contains a reference to a `Vertex` object, the other end of an edge.

Edge annotations were encoded as a extensions — i.e., extra fields — of the `Neighbor` class. Figure 3 illustrates this representation. By pushing the neighbor `Vertex` object and edge annotations into a `Neighbor` object, we reduced the number of list accesses required to obtain these annotations. While this did lead to a simplification of the coding of some mixin-layers, it did not simplify the complexity of the Kruskal algorithm. Since this mixin-layer was unnecessarily difficult to write (and read!), we knew there was still something wrong. This lead to our final design.

4.3 Edge-Neighbor Representation (GEN)

Textbook descriptions of algorithms are almost always simple. The reason is that certain implementation details have been abstracted away — but this is, in fact, the strength of layers and extensions. We wanted to demonstrate that we could (almost literally) copy algorithms directly out of text books into mixin-layer code. The benefits of doing so are (a) faster and more reliable implementations and (b) easier transference of proofs of algorithm correctness into proofs of program correctness. We realized that the only way this was possible was to recognize that there are a standard set of "conceptual" objects that are referenced by all graph algorithms: Graphs, Vertices, Edges, and Neighbors (i.e., adjacencies). Algorithms in graph textbooks define the fundamental extensions of graphs, and these extensions modify Graph objects, Vertex objects, Edge objects, and Neighbor objects. Thus, the simplest way to express such extensions is to reify all of these "conceptual" objects as physical objects and give them their own distinct classes.

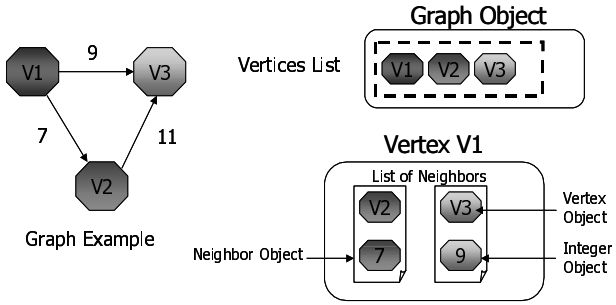


Figure 3. Neighbor Lists Representation Example

The problems of our previous designs surfaced because we tried to make "short-cuts" to avoid the explicit representation of certain conceptual objects (e.g., Edge, Neighbor). Our justification for doing so was because we felt the resulting programs would be more efficient. That is, we were performing "optimizations" in our earlier designs that folded multiple conceptual objects into a single physical object. In fact, such *premature optimizations* caused us nothing but headaches as we tried to augment our design to handle new extensions and to produce easy to read and maintain code. (We think that this may be a common mistake in most software designs, not just ours). So our "final" design made explicit all classes of objects that could be explicitly extended by graph algorithms. Namely, we had four classes:

- `Graph`: contains a list of `Vertex` objects, and a list of `Edge` objects.
- `Vertex`: contains a list of `Neighbor` objects.
- `Neighbor`: contains a reference to a neighbor `Vertex` object (the vertex in the other end of the edge), and a reference to the corresponding `Edge` object.
- `Edge`: extends the `Neighbor` class and contains the start `Vertex` of an `Edge`.

Edge annotations are now expressed as extensions of `Edge` class, and were expressed by the addition of extra fields in the `Edge` class. This representation is illustrated in Figure 4.

Equating conceptual objects with physical objects may simplify source code, but the question remains: were our original designs more efficient? Is "premature design optimization" essential for performance? These questions are addressed next.

5 Profiling Results

We performed a series of benchmarks to quantify the trade-offs between our three designs. Several implementations of the designs were tried, using different containers, and different strategies to access and copy the edge annotations. This section shows the results for our most fine-tuned implementations. The benchmarks were run on a Windows 2000 platform using a 700Mhz processor with 196MB RAM.

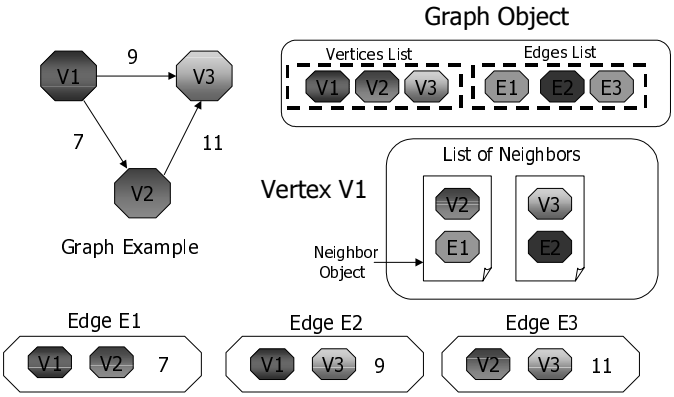
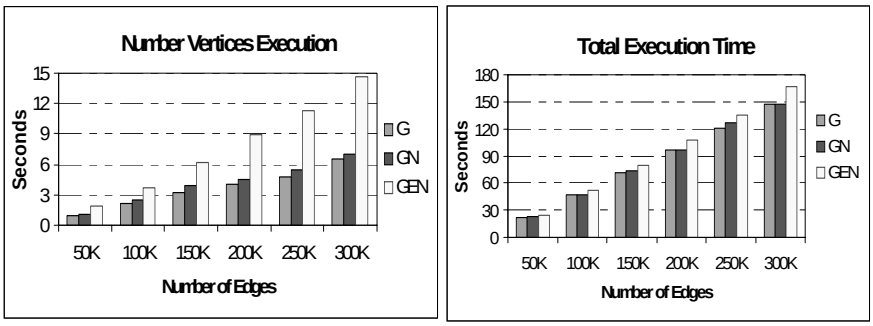


Figure 4. Edge and Neighbor List Representation Example

The first program used the vertex number algorithm on undirected graphs using depth first search. This program measured the performance of graph creation and traversal. A randomly generated graph with 1000 vertices was used as test case. Figure 5 shows the benchmark results.

Figure 5a indicates that design **G** (our first) performs better than the other two; 6%-22% better that **GN** (our second), and 75%-120% better than **GEN** (our third). This is not surprising: **GN** and **GEN** have object creation overhead that is absent in **G** — `Neighbor` objects are created in **GN**, and `Neighbor` and `Edge` objects are created in **GEN**. While this is an obvious difference, the overall speed of the benchmark was dictated by the time reading the graph from disk. Figure 5b shows this total execution time, where the difference between the **G** application and the **GN** application is about 5% and **G** with **GEN** is about 9%.

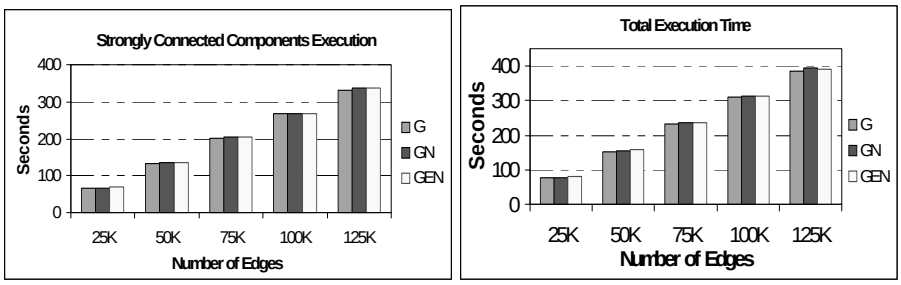


(a) (b)

Figure 5. Simple graph traversal comparison

The second program benchmarked the impact of copying a graph with edge annotations. `StronglyConnected` utilizes such an operation, transpose, that creates a new copy of a graph but with the direction of the edges reversed. A randomly generated

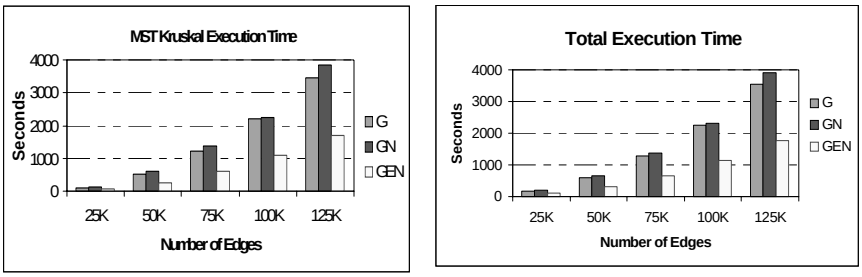
graph with 500 vertices was used as test case. In general, there was no significant difference (see Figure 6a). The **G** design performed 2% better than **GN** and 6% better than **GEN**. Although cost of graph creation is different among designs (as indicated by Figure 5a), the differences are swamped by the large computation times of the `StronglyConnected` algorithm. In particular, only 15% of the total execution time in Figure 6b was spent in reading the graph in from disk.



(a) (b)

Figure 6. Strongly Connected Components

The third program benchmarked the impact of algorithms that use edges explicitly, like Kruskal's algorithm. A randomly generated graph with 500 vertices was used as a test case. As expected, the **GEN** representation outperformed the other two simply because it does not have to compute and create the edges from the adjacency or neighbor lists. It performed between 43% and 98% faster than representation **G**, and between 59% and 120% faster than representation **GN** (see Figure 7a). The difference between **G** and **GN** is due to the fact that in the latter, to get the weights for each edge, an extra access to the weights lists is required; and that the creation of the output graph is more expensive because it has to create `Neighbor` objects as well. Of the total execution time (Figure 7b), approximately 60% was spent reading a graph of 25K edges from disk, and less than 5% when the graph had 125K edges.



(a) (b)

Figure 7. MST Kruskal

Overall, we found that the performance of algorithms that did not use weighted edges (e.g., numbering, cycle-checking, connected components, strongly-connected compo-

nents) had slightly better performance with the **G** design. For those algorithms that used weighted edges (e.g., MST Prim, MST Kruskal, shortest path), the **GEN** design was better. Because an application is specified by the same equation for all three models, we could exploit our performance observations in a "smarter" generator that would decide which design/implementation would be best for a particular family member — i.e., one equation might be realized by a **G** design, another by a **GEN** design (see [6]).

Focussing exclusively on performance may be appropriate for most applications. But a more balanced viewpoint needs to consider program complexity (which indirectly measures the ease of maintenance, understandability, and extensibility). The main issue for us was the impact that the representation of edges had on program complexity. By in large, all layers had visually simple representations. But the Kruskal layer seemed more complicated than it needed to be. The reason was that in both the **G** and **GN** designs, the Kruskal layer had an explicit `Edge` class that was private to that layer, and used by no other layer[4]. (The Kruskal algorithm demanded the existence of explicit edge objects). The fact that all layers might benefit from making `Edge` explicit drove us to the **GEN** design, which we considered visually and conceptually more elegant than our earlier designs. As it turns out, our instincts on "visual simplicity" were not altogether accurate. To see why, we use two metrics for program complexity: *lines of code* (**LOC**) and *number of symbols* (**NSYMB**).[5] Table 3 shows these statistics for the Kruskal layer. Making edges explicit did indeed simplify this layer's encoding. However, other parts of our design grew a bit larger (mostly because we had to make the `Neighbor` and `Edge` classes and their extensions explicit). Table 4 shows these same statistics, across all layers, for all three designs. Overall, the statistical complexity of all three designs was virtually identical. So the drive for "visual simplicity" among layers in the end did improve our designs, but surprisingly did not impact their size statistics.

There is a benefit to the **GEN** design that is not indicated by the above tables. If we chose to enlarge the **G** and **GN** product-line with more algorithms that directly manipulate edges, then it is likely a local copy of the `Edge` class would be introduced into these layers. And doing so would result in replicated code, possibly leading to problems with program maintenance. By making the `Edge` class global to all extensions as in the **GEN** design, we would expect little or no code replication — precisely what we want in a product-line design.

Finally, we wanted to see if explicit layering (which mixin-layers produce) affects the overall performance. We created equations for each design that contained the most layers (10), and manually-inlined the resulting chain of mixin-layers into an unlayered package called *Flat*. There are two equations that have 10 layers, namely:

---

4. The local version of Edge in the Kruskal layer is indicated in Table 4 as 7 lines of 52 tokens.
5. We used other metrics [8], but found they provided no further insights.

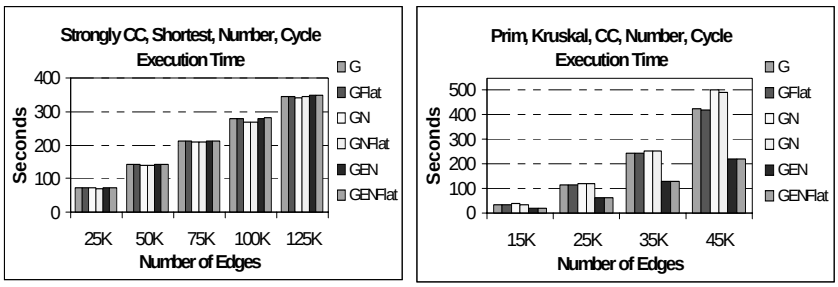
	G	LOC GN	GEN	G	NSYMB GN	GEN
Kruskal	**87**	**90**	**69**	**927**	**928**	**695**

Table 3. Kruskal Algorithm Statistics

Class Name	G	LOC GN	GEN	G	NSYMB GN	GEN
Graph	372	387	380	3554	3600	3492
Vertex	209	202	198	1832	1758	1631
Neighbor	0	30	16	0	229	49
Edge	7	7	26	52	52	304
Total	**588**	**626**	**620**	**5438**	**5639**	**5476**

Table 4. Lines of Code (LOC) and Number of Symbols (NSYMB)

- *Directed, Weighted, DFS, StronglyConnected, Number, Transpose, Shortest, Cycle, Benchmark, Prog:* in this case the difference between the layered version and the flattened one oscillates between 0% and 2% in **G**, -1% and 1% for **GN**, and -1% and 1% for **GEN**. A randomly generated graph with 500 vertices was used as test case. These results are shown in Figure 8a.
- *Undirected, Weighted, DFS, Connected, Number, Cycle, MST-Kruskal, MST-Prim, Benchmark, Prog:* for this application the difference between the layered version and the flattened one varies between 0% and 3% in **G**, 0% and 5% in **GN**, and between -1% and 1% in **GEN**. A randomly generated graph with 300 was used as test case. The results are shown in Figure 8b.



(a) (b)

Figure 8. Effect of Class Layering

The small difference between the layered version and its corresponding flattened one is due to the fact that few methods override their parent method. When overriding does occur, it involves fewer than 3 layers. Again, this result is specific to GPL and may not hold for other domains.

6 Conclusions

GPL is a simple and illustrative problem for product-line designs. Different applications of the GPL product-line are defined by unique sets of features, and not all combinations of features are permitted. The state of the art in product-lines is immature, and the need to understand the commonalities and differences among product-line design methodologies is important. We want to know how methodologies differ, what are their relative strengths and weaknesses, and most importantly what are the scientific principles that underlie these models. We do not know answers to these questions. But it is our belief that by proposing and then solving a standard set of problems, the answers to these questions will, in time, be revealed.

We believe GPL is a good candidate for a standard problem. It has the advantages of *simplicity* — it is an exercise in design and implementation that can be discussed in a relatively compact paper — and *understandability* — domain expertise is easily acquired because it is a fundamental topic in Computer Science. Further, it provides an interesting set of challenges that should clearly expose the key concepts of product-line methodologies.

In this paper, we presented a product-line model and implementation of GPL using the GenVoca methodology and the Jakarta Tool Suite (JTS). We showed how GenVoca layers correspond to features, and how compositions of features are expressed by equations implemented as inheritance lattices. We presented a sequence of designs that progressively simplified layer implementations. We benchmarked these implementations to understand performance trade-offs. As expected, different designs do have different execution efficiencies, but it is clear that a "smart" generator (which had all three designs available) could decide which representation would be best for a particular application. As an additional result, we showed that there is a very small impact of class layering in overall application performance.

We hope that others apply their methodology to GPL and publish their designs and findings. We believe that our work would benefit by a close inspection of others, and the same would hold for other methodologies as well. Our code can be downloaded from `http://www.cs.utexas.edu/users/dsb/GPL.html`.

Acknowledgements. We would like to thank Vijaya Ramachandran for her valuable help with the subtle details of the theory of graph algorithms. We also thank Jay Misra for clarifying the distinction between refinements and extensions.

7 References

[1] J-R Abrial, E. Boerger, and H. Langmaack, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, Lecture Notes in Computer Science, Vol. 1165, Springer-Verlag, 1996.

[2] P. America, et. al. "CoPAM: A Component-Oriented Platform Architecting Method Family for Product Family Engineering", *Software Product Lines: Experience and Research Directions*, Kluwer Academic Publishers, 2000.

[3] D. Batory and B. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*, February 1997.

[4] D.Batory, B.Lofaso, and Y.Smaragdakis. "JTS: Tools for implementing Domain-Specific Languages", *Int. Conf. on Software Reuse*, Victoria, Canada, June 1998.

[5] D. Batory, R. Cardone, and Y.Smaragdakis. "Object-Oriented Frameworks and Product Lines", *1st Software Product-Line Conference*, Denver, Colorado, August 2000.

[6] D. Batory, G. Chen, E. Robertson, and T. Wang, "Design Wizards and Visual Programming Environments for GenVoca Generators", *IEEE Transactions on Software Engineering*, May 2000, 441-452.

[7] J. Bosch, "Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study", *Software Architecture*, Kluwer Academic Publishers, 1999.

[8] S.R. Chidamber and C.F. Kemerer, "Towards a Metrics Suite for Object Oriented Design", *OOPSLA 1991*.

[9] S. Cohen and L. Northrop, "Object-Oriented Technology and Domain Analysis", *Int. Conf. on Software Reuse*, Victoria, Canada, June 1998.

[10] T.H. Cormen, C.E. Leiserson, and R.L.Rivest. *Introduction to Algorithms*, MIT Press, 1990.

[11] K. Czarnecki and U.W. Eisenecker, "Components and Generative Programming", *SIGSOFT 1999*, LNCS 1687, Springer-Verlag, 1999.

[12] K. Czarnecki and U.W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[13] J-M. DeBaud and K. Schmid, "A Systematic Approach to Derive the Scope of Software Product Lines", *Int. Conference on Software Engineering 1999*.

[14] H. Gomaa et al., "A Prototype Domain Modeling Environment for Reusable Software Architectures", *Int. Conf. on Software Reuse*, Rio de Janeiro, November 1994, 74-83.

[15] M. Griss, "Implementing Product-Line Features by Composing Component Aspects", *First International Software Product-Line Conference*, Denver, Colorado., August 2000.

[16] I. Holland. "Specifying Reusable Components Using Contracts", *ECOOP 1992*.

[17] D.L. Parnas, "On the Design and Development of Program Families", *IEEE Transactions on Software Engineering*, March 1976.

[18] Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers", *ECOOP 1998*.

[19] M. VanHilst and D. Notkin, "Using C++ Templates to Implement Role-Based Designs", *JSSST International Symposium on Object Technologies for Advanced Software*, Springer-Verlag, 1996, 22-37.

[20] D.M. Weiss and C.T.R. Lai, *Software Product-Line Engineering*, Addison-Wesley, 1999.